System-Level Safety and Certification Implications of Using Linux in Airborne Avionics

Haoran Lu *

Posted Date: 10 April 2026

doi: 10.20944/preprints202603.0354.v6

Keywords: airworthiness; DO-178C; DO-330; determinism; isolation; Linux; operating system certification; partition; trusted computing base; execution semantics

 doi:10.20944/preprints202603.0354.v6

Article

System-Level Safety and Certification Implications of Using Linux in Airborne Avionics

Haoran Lu

Symthosm, Shanghai, China

Abstract

This paper presents a certification-oriented, system-level analysis of using Linux in safety-critical airborne avionics, with emphasis on Design Assurance Level (DAL) A/B systems. Linux is a feature-rich general-purpose OS whose open and dynamic execution semantics can be difficult to finitely bound and operationally "freeze" at integration time. We analyze how key architectural characteristics of Linux—including a large trusted computing base (TCB), asynchronous kernel activity, mutable memory mappings, monolithic privilege domains, and a rapidly evolving toolchain—interact with assurance objectives commonly expected under DO-178C and DO-330. The analysis identifies eight independently sufficient certification-relevant risk factors affecting temporal determinism, spatial isolation, fault containment, configuration stability, and lifecycle assurance feasibility. To avoid fragmented observations, these factors are consolidated into a unified causal framework that traces certification challenges back to two consequence categories: airworthiness feasibility constraints and semantic complexity. The framework also evaluates commonly proposed mitigations (e.g., PREEMPT_RT, containers, and static configuration) and explains why these measures may not address the underlying system-level issues. The contribution of this work is a structured argumentation framework that makes architectural implications explicit and supports operating-system selection and safety governance decisions in integrated modular avionics.

Keywords: airworthiness; DO-178C; DO-330; determinism; isolation; Linux; operating system certification; partition; trusted computing base; execution semantics

1. Introduction

Over the past decade, the global aviation industry has entered a phase of steady and diversified expansion, driven simultaneously by the modernization of traditional commercial fleets and the rapid emergence of low-altitude economic activities. New operational domains—such as unmanned aerial systems, urban air mobility, electric vertical-takeoff-and-landing (eVTOL) vehicles, and low-altitude logistics—have significantly broadened the aviation technology landscape. In China in particular, national and regional policies supporting low-altitude airspace reform have accelerated industrial development and attracted substantial private-sector investment. For this reason, the ecosystem now consists not only of established aerospace manufacturers but also of a large number of newly formed or cross-industry entrants whose engineering experiences originate from consumer electronics, robotics, and commercial embedded-systems sectors rather than from safety-critical aviation.

This shift in industry demographics creates a structural divergence in system-architecture choices. Well-resourced organizations—especially those with prior experience in safety-critical projects—typically adopt mature, certifiable real-time operating systems (RTOSs) such as VxWorks 653 or PikeOS. These platforms are explicitly designed around partitioning principles and provide the deterministic behavior, analyzable execution semantics, and long-term configuration stability required under DO-178C. Their high licensing and integration costs are acceptable for organizations whose safety processes, program governance, and airworthiness culture already align with traditional avionics expectations. [3,21,23]

 doi:10.20944/preprints202603.0354.v6

However, a large and rapidly growing group of resource-constrained new entrants faces a different set of incentives. Lacking deep familiarity with aviation certification and seeking to minimize time-to-market and platform cost, these companies increasingly turn to Linux—an open-source, feature-rich, and widely supported operating system. Linux offers obvious practical advantages: extensive driver availability, mature tooling, broad developer familiarity, and zero licensing fees. For commercial embedded markets, these attributes legitimately accelerate prototyping and reduce development cost. For teams unfamiliar with DO-178C, Linux may appear not only economical but also technologically "good enough", especially when combined with hardening efforts or real-time extensions such as PREEMPT_RT. [3,30]

Furthermore, some international Linux institutions are studying the potential role of open-source platforms in future avionics architectures. This has contributed to a widespread misconception, particularly in emerging low-altitude aviation sectors.

This paper addresses that misconception from a system-safety and certification perspective. By examining Linux through the lens of DO-178C/DO-330 lifecycle assurance objectives, we show that Linux's open-world execution semantics, timing variability, mutable memory mappings, monolithic trusted computing base (TCB), and continuously evolving toolchain can be misaligned with the architectural assumptions typically used to achieve certifiable airborne platforms at DAL A/B. We present a certification-oriented, system-level analysis that links Linux's architectural properties to concrete assurance impacts and synthesizes them into a unified causal framework. The framework provides a structured basis for operating-system selection, technology governance, and lifecycle planning.

2. Related Work

Research on operating system (OS) suitability for safety-critical avionics has long emphasized the need for deterministic execution, strong spatial and temporal isolation, and a verifiable and stable software baseline. The ARINC 653 family of standards formalizes these requirements through a partitioned execution model with fixed time windows, memory protection regions, and a minimal, analyzable separation kernel architecture. Meanwhile, airworthiness guidance such as RTCA DO-178C primarily defines lifecycle assurance objectives (requirements traceability, verification rigor, and configuration control) rather than prescribing a specific partitioned runtime model. In practice, certifiable avionics platforms are therefore expected to exhibit closed, finitely analyzable execution semantics at integration time—an expectation that general-purpose operating systems typically cannot satisfy. [1–3]

In contrast to separation kernels, monolithic general-purpose kernels such as Linux implement rich functionality and dynamically evolving subsystems whose behavior depends heavily on runtime conditions. Prior work on Linux real-time extensions, particularly PREEMPT_RT, has focused on reducing kernel latency through techniques such as threaded interrupt handlers and priority-inheritance locks, and these developments have recently culminated in upstream integration of PREEMPT_RT into mainline kernels. However, the PREEMPT_RT project itself acknowledges ongoing challenges with non-preemptible code paths, memory-management latency, and concurrent subsystem interactions that can complicate deterministic timing guarantees. [30]

In parallel, the avionics research community has developed partitioning hypervisors and separation kernels as a more certifiable alternative to monolithic kernels. XtratuM, for example, implements strong temporal and spatial partitioning through a minimal hypervisor architecture designed for real-time embedded systems, enabling deterministic scheduling and strict fault containment boundaries consistent with ARINC 653 principles. [1,13]

More recently, industry initiatives such as the Enabling Linux in Safety Applications project (ELISA) have sought to define processes, tools, and artifacts supporting the evaluation of Linux-based systems in safety-related domains. ELISA collaborates with certification bodies and industrial stakeholders and maintains working groups for sectors including aerospace. However, ELISA explicitly states that it cannot produce a certifiable Linux kernel or guarantee compliance with

 doi:10.20944/preprints202603.0354.v6

aviation standards, positioning its outputs as exploratory guidance rather than certification-oriented evidence. Accordingly, while ELISA demonstrates increasing industrial interest in adapting Linux to safety-related use cases, it does not alter the fundamental architectural considerations underlying DO-178C and ARINC 653 compliance. [36]

Collectively, existing work highlights a clear divide between certifiable partitioned architectures and the open-world semantics characteristic of general-purpose kernels like Linux. While real-time patches and safety-focused initiatives improve aspects of the Linux ecosystem, they do not eliminate the architectural mismatches that underpin airworthiness challenges for Design Assurance Level (DAL) A/B avionics systems.

3. Certification-Oriented System-Level Analysis Framework

This work proposes a certification-oriented, system-level analysis framework grounded in standards-based reasoning and architectural semantics rather than empirical benchmarking or prototype construction. The framework is intended to be transferable: it can be applied to other candidate operating-system architectures by mapping documented architectural properties to assurance objectives and identifying system-level risk factors for DAL A/B certification under DO-178C and DO-330.

The analysis proceeds in three structured and mutually reinforcing stages:

3.1. Characterization of Linux Architectural Semantics

The first stage systematically characterizes Linux's architectural semantics—its monolithic kernel structure, dynamic memory-management behavior, asynchronous subsystem interactions, and other properties documented in the real-time Linux and kernel-execution literature.[25,27,30]

3.2. Identification of Independently Sufficient Certification-Relevant Risk Factors

These architectural characteristics are systematically mapped against DAL A/B assurance objectives as commonly interpreted in certification practice. This standards-based mapping yields eight independently sufficient certification-relevant risk factors. Any one of these factors can, on its own, undermine key DO-178C/DO-330 objectives (e.g., determinism, isolation, fault containment, configuration stability, or lifecycle assurance feasibility). Considered together, the eight risk factors reveal the breadth of challenges that arise when attempting to use Linux as a certification baseline for high-integrity airborne systems.

3.3. Unified Causal Framework and Examination of Mitigation Narratives

To avoid treating these eight risk factors as isolated observations, the analysis consolidates them into a unified causal structure. This structure traces the risk factors back to a shared architectural root cause—Linux's open-world, general-purpose design—and two consequence categories: airworthiness feasibility constraints and semantic complexity. To reduce ambiguity in interpretation, the paper also examines commonly proposed mitigation narratives (e.g., PREEMPT_RT, containers, and static configuration) and explains, within the same causal structure, why these measures may not address the underlying system-level issues.

This methodology relies exclusively on openly documented architectural behaviors and publicly available certification requirements. It constructs an evidence-based, standards-aligned, system-level certifiability argument, consistent with established analytical practices in safety-critical systems engineering.

 doi:10.20944/preprints202603.0354.v6

4. Certification-Relevant Architectural Risk Factors of Linux in Safety-Critical Avionics

The certification challenges associated with Linux-based avionics can be traced to eight architectural risk factors, each of which can independently undermine assurance objectives expected for DAL A/B airborne systems (e.g., under DO-178C/DO-330 practice). Each factor is analyzed below and contrasted with design expectations commonly adopted in avionics-grade partitioned platforms.

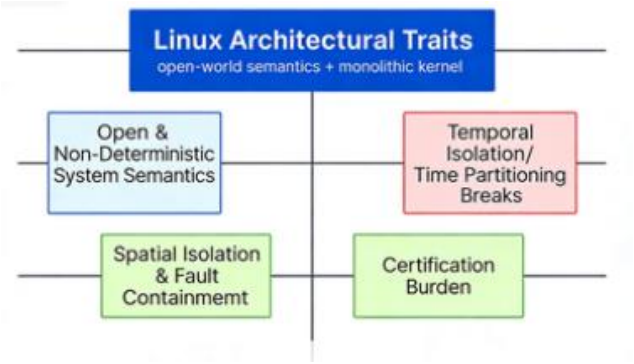


Figure 1. Classification of key Linux architectural traits.

4.1. Open Execution Semantics with Unpredictable Behavior

Linux exhibits inherent open-world system semantics, characterized by a vast, dynamically evolving set of execution paths whose behavior is shaped by runtime operating conditions, workload characteristics, and autonomous interactions among subsystems. These execution paths arise from diverse kernel-internal mechanisms—including dynamic memory-management activities, interrupt and deferred-work handling, and subsystem-specific state transitions—which evolve independently of application logic and cannot be enumerated or frozen at integration time. The combined effect of these mechanisms results in an unbounded, non-enumerable system state space, driven by hardware events, concurrent execution, dynamic memory pressure, and load-dependent scheduling, which defies closed-form formal analysis and static verification. This core design attribute conflicts with the DO-178C assurance model for DAL A/B systems, which depends on a defined, reviewable integrated baseline and on the ability to specify, verify, and re-verify intended behavior (including timing and partitioning assumptions) against that baseline. When execution paths and interference patterns are created by runtime conditions rather than by requirements-defined design, comprehensive verification closure, bounded timing evidence, and demonstrable partitioning assurances cannot be sustained. Linux's semantic openness is therefore the primary cause of its subsequent temporal and spatial-isolation failures, because it inherently precludes the static, bounded, and fully verifiable behavior expected of high-integrity DAL A/B avionics systems. [1]

Linux further increases run-to-run variability through dynamic linking and runtime loader behavior. Library composition and symbol resolution can vary with the environment and loader configuration, so the effective executable content may differ across deployments.

 doi:10.20944/preprints202603.0354.v6

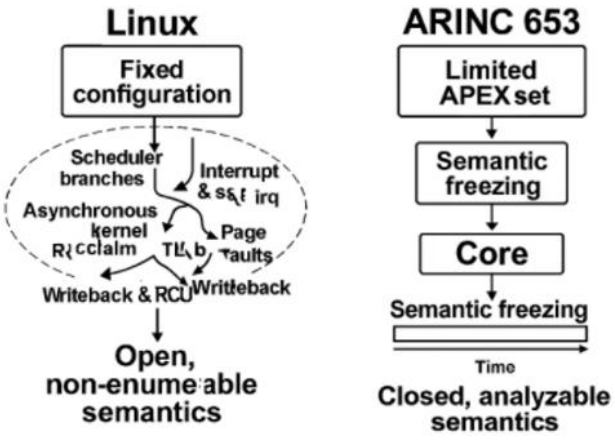


Figure 2. Linux semantic unfreezing vs ARINC 653 semantic freezing.

In summary, Linux's open-world semantics makes the integrated behavior difficult to bound, review, and reproduce at certification baselines, and this non-enumerability directly feeds into the temporal and spatial isolation problems analyzed in Sections 4.2 and 4.3.

4.2. Lack of Temporal Determinism

DO-178C Section 2.4.1(b) describes that when multiple software components share a computing resource, partitioning is needed to prevent adverse interactions, including timing interference. In certification practice this translates into the need for analyzable upper bounds on component release latency, dispatch delay, kernel service time, and interference from other software components. The Linux process–thread execution model cannot supply such guarantees because scheduling decisions, interrupt handling, and kernel activity are driven by dynamically evolving workload, subsystem state, and asynchronous events. As a result, neither processes nor threads can be associated with statically provable determinism time. [3]

In Linux, temporal nondeterminism arises from two architecturally distinct mechanisms whose effects on execution timing must be analyzed separately.

4.2.1. Scheduler-Driven Nondeterminism

Priority-based scheduling in Linux does not guarantee immediate execution for a runnable high-priority task because scheduling behavior depends on the dynamic state of the system, including run-queue composition, wakeup patterns, migration decisions, and internal kernel bookkeeping. These behaviors occur even in the absence of external events.

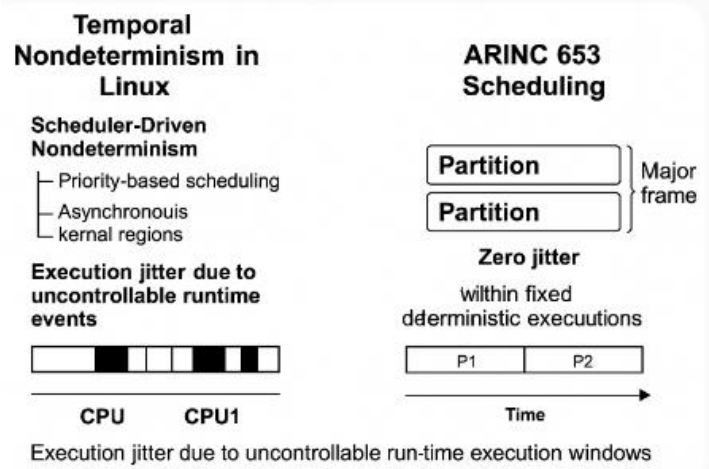


Figure 3. Scheduler-driven execution jitter in Linux vs deterministic ARINC 653 partition.

 doi:10.20944/preprints202603.0354.v6

Furthermore, the Linux kernel contains numerous non-preemptible regions, such as:

- spinlock-protected critical sections,
- per-CPU data updates,
- scheduler state transitions,
- low-level exception-handling paths.

While these sections execute, preemption is explicitly disabled, preventing the scheduler from dispatching a higher-priority process or thread until the critical section completes. The duration of these regions is runtime-dependent and influenced by cache state, lock contention, and microarchitectural factors, making their execution time analytically unbounded.

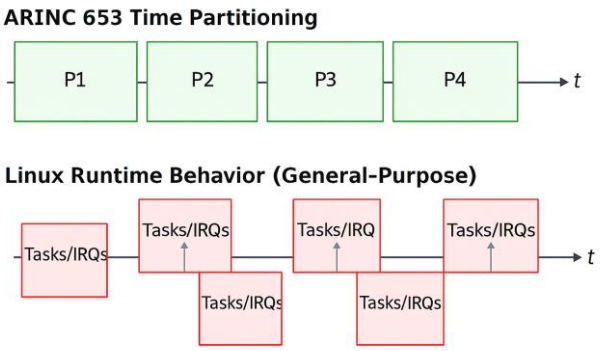


Figure 4. Event-driven execution jitter in Linux vs ARINC 653 fixed time partitions.

4.2.2. Event-Driven (Asynchronous) Nondeterminism

Separate from scheduler-driven effects, Linux also contains multiple asynchronous execution sources—including hardware interrupts, software interrupt request (softirq) processing, network-stack activity, timer callbacks and memory-reclaim operations. These activities are triggered by external stimuli or internal system conditions and may occur at arbitrary times. As such, they can preempt a running task immediately or occupy CPU time through deferred work, introducing additional timing variability that cannot be bounded statically.

Asynchronous nondeterminism therefore reflects the open-world, event-driven nature of the kernel's interaction with devices, resource pressure, and subsystem events—factors inherently outside the scheduler's deterministic control.

Conversely, ARINC 653-compliant platforms use a predefined major/minor-frame scheduling model with fixed, deterministic execution windows for all subsystems, eliminating runtime jitter from unconstrained events.

4.3. No Physical Memory Isolation

DO-178C Section 2.4 and 2.4.1 notes that partitioning may be achieved by allocating unique hardware resources to each software component (that is, only one software component is executed on each hardware platform in a system). Alternatively, partitioning provisions may be made to allow multiple software components to run on the same hardware platform. Interpreting the first approach, placing components on different hardware platforms effectively creates distinct physical resource domains, eliminating shared-memory interaction by construction. For the second approach (co-resident components on the same platform), partitioning must still ensure that one component is not allowed to corrupt or modify another component's code, input/output (I/O), or data storage areas; in practice this requires demonstrable protection of memory and storage resources via hardware and/or a combination of hardware and software, as described in the standard. However, Linux relies on a

 doi:10.20944/preprints202603.0354.v6

dynamic and mutable virtual-memory architecture that violates these requirements. Several core mechanisms illustrate this behavior:

- Demand paging and on-demand allocation. Page tables are populated lazily, and physical pages may be allocated or remapped during runtime based on memory pressure and process behavior.
- Page reclaim and compaction. Under memory pressure, the kernel evicts or relocates physical pages, invoking reclaim, compaction, or write-back paths that modify page-table mappings without application involvement.
- Dynamic page-table updates and Translation Lookaside Buffer (TLB) shootdowns. Linux frequently updates page attributes, permission bits, and mapping structures, triggering cross-CPU TLB invalidations and modifying the effective physical-memory layout during operation.

Because Linux performs these modifications autonomously, a process or thread cannot be associated with a fixed, statically provable physical-memory region. No mechanism prevents the kernel from reassigning or altering physical pages that lie within or adjacent to the memory range used by a safety-critical application, nor does Linux provide hardware-enforced barriers preventing other components from accessing or corrupting those regions.

Thus, Linux's dynamic page-table management conflicts with the DO-178C requirement that a partition's physical memory be statically allocated, hardware-isolated, and invariant throughout system operation.

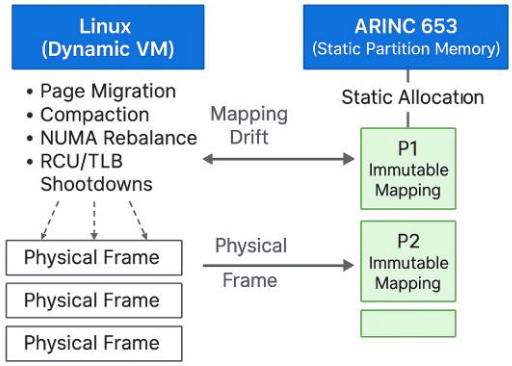


Figure 5. Linux memory mapping drift vs ARINC 653 fixed memory mapping.

4.4. Lack of Fault Isolation

Linux does not provide certifiable inter-process fault containment. All user processes ultimately depend on a single, shared, monolithic kernel, and this kernel is responsible for handling every system call. Because all processes share the same privileged kernel address space and kernel-resident global structures, a fault in any process can corrupt kernel state that is globally visible and globally trusted.

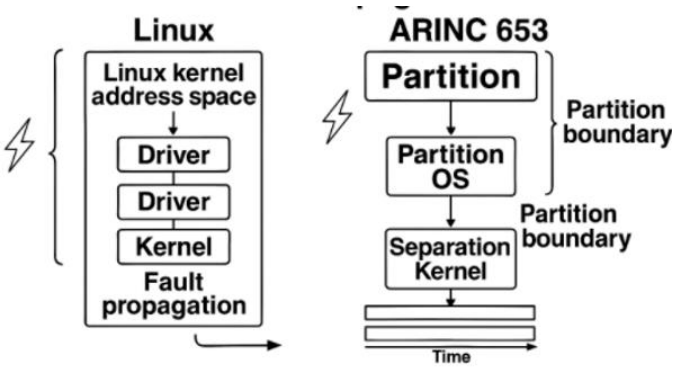


Figure 6. Linux monolithic kernel with drivers vs ARINC 653 separation kernel.

 doi:10.20944/preprints202603.0354.v6

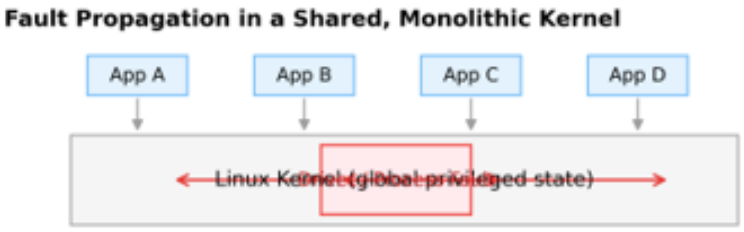


Figure 7. Fault propagation in Linux.

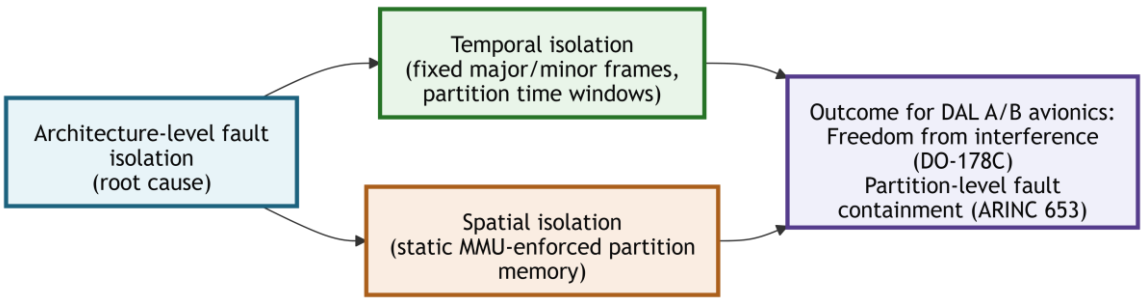
In practice, this means that a defect in one component may propagate through:

- shared kernel memory regions used by subsystems such as the scheduler, virtual file system (VFS), networking, and memory management;
- global locks and synchronization primitives that serialize access across unrelated components;
- reference counters and object life-cycle structures (e.g., file descriptors, network buffers, slab objects);
- interrupt-handling and softirq pathways, whose execution contexts are shared across all processes regardless of their criticality.
- shared and dynamically managed physical-memory pools and page-table state, rather than fixed, non-overlapping, MMU-enforced partition memory regions;
- non-partitioned, best-effort CPU scheduling (ASAP), rather than a table-driven major/minor-frame schedule with predefined execution windows.

Since these structures are not partition-scoped—and cannot be restricted or made private to individual processes—Linux cannot prevent a fault originating in one process from affecting the execution correctness, timing, or stability of others. This propagation mechanism is inherent to monolithic-kernel design and directly contradicts the fault isolation defined for DAL A/B airborne software under DO-178C Section 2.4.1(c), including the need to avoid adverse effects across partitioned components.

Commercial avionics RTOS products implementing ARINC 653 adopt the opposite model. They enforce strict partition boundaries using:

- hardware Memory Management Unit (MMU) isolation with statically defined, non-overlapping physical memory regions;
- a minimal, rigorously verified separation kernel responsible only for scheduling partitions and mediating controlled IPC;
- complete disallowance of shared kernel-writable global state between partitions;
- static temporal partitioning with fixed, predefined time windows to guarantee CPU execution isolation between partitions;
- hierarchical health monitoring (HM) mechanisms supporting fault detection, containment, and recovery at process, partition, and system levels;
- fault-containment boundaries that ensure a failure inside one partition cannot corrupt the separation kernel or any other partition.



 doi:10.20944/preprints202603.0354.v6

Figure 8. Relationship among architecture-level fault isolation and temporal/spatial isolation for DO-178C DAL A/B.

As such, ARINC 653 systems achieve true inter-partition fault isolation, with failures strictly contained to the originating partition. This property is fundamental to satisfying DO-178C fault-propagation analysis for DAL A/B functions.

4.5. Driver Contamination of Kernel Global State

Linux device drivers execute with full kernel privilege, sharing the monolithic kernel's global address space. Eventually, a single defective driver can inadvertently corrupt system-wide shared state, including the structures mentioned in the above sections. Unlike the previous section, where a partially faulty user component may only affect others indirectly by first corrupting kernel state through constrained interfaces, an in-kernel driver can directly write to global kernel memory and thereby directly compromise unrelated components across the system.

Because these structures are global, faults originating in one driver can propagate across unrelated components, undermining the system's ability to isolate erroneous behavior. This propagation pathway is incompatible with avionics fault-containment principles, where failures within one component must not compromise the integrity or availability of others.

By contrast, INTEGRITY-178B is commonly deployed with a separation architecture that keeps the certified kernel small and avoids executing device drivers inside the privileged kernel address space. When drivers are placed outside the kernel as separate components, they no longer share the kernel's writable global data structures; on this account, a driver defect is less likely to directly corrupt kernel-internal objects or silently poison system-wide kernel state. This sharply contrasts with Linux's monolithic in-kernel driver model, where drivers and core subsystems inherently reside in and mutate the same global kernel memory.

4.6. Overly Large Trusted Computing Base (TCB)

Linux integrates millions of lines of kernel code spanning diverse and continuously evolving subsystems, including device drivers, networking stacks, filesystems, IPC frameworks, tracing and debugging infrastructures, and numerous optional kernel features, all resident components form part of the Trusted Computing Base (TCB). Under DO-178C, every TCB-resident element contributes directly to certification scope and must undergo full lifecycle assurance activities. For a codebase of this scale and complexity, the resulting verification burden becomes economically prohibitive and technically impractical.

A breakdown of this burden along the five major DO-178C process domains illustrates the mismatch.

4.6.1. Planning Process Impact (PSAC, Standards, Objectives Allocation)

Certification begins with the software planning process and the Plan for Software Aspects of Certification (PSAC). Under DO-178C Section 4, planning must define the software life cycle and its inter-process relationships, select and define the software life cycle environment (methods and tools), and produce software plans/standards that are placed under change control and reviewed (DO-178C 4.1, 4.2, 4.3, 4.2(g), and 4.6). For DAL A/B, this implies that every software component contributing to safety objectives—and the means used to develop, verify, build, and load it—must be explicitly identified and governed by the plans. For Linux, this would require:

- Declaring all kernel subsystems, bundled drivers, and architecture-specific code paths that can affect safety objectives as part of the certifiable baseline, and assigning appropriate DAL driven objectives in the PSAC.
- Producing planning artifacts (PSAC, SDP, SVP, SCMP, SQAP) that define the software life cycle, transition criteria, and feedback mechanisms, and that also define the software life cycle

 doi:10.20944/preprints202603.0354.v6

environment (methods/tools) used to develop, verify, build, and load the kernel—placing these plans and standards under change control and review (DO-178C Section 4).

- Defining and maintaining an assurance strategy for kernel-wide concurrency, shared memory, interrupts, scheduling, and dynamic allocation, while also controlling the development environment (compiler/linker/loader versions and options). DO-178C notes that introducing new compiler/linker/loader versions or changing options may invalidate prior tests and coverage and requires planned re-verification means (Section 4.4.2(c)).

Because Linux includes thousands of modules and hundreds of interdependent subsystems, planning artifacts would become unmanageably large, and many required DAL A/B planning commitments (e.g., complete design traceability or determinism justification) simply cannot be demonstrated.

4.6.2. Development Process Impact (Requirements, Design, Code)

DO-178C Section 5 defines a requirements-driven development chain in which high-level requirements (HLR) are developed from system inputs (DO-178C 5.1), refined into a software architecture and low-level requirements (LLR) suitable for direct coding (DO-178C 5.2), implemented as source code from those low-level requirements (DO-178C 5.3), and integrated into executable object code using controlled compiling/linking/loading data (DO-178C 5.4). DO-178C further requires bi-directional traceability across system requirements, high-level requirements, low-level requirements, and source code (DO-178C 5.5), so that completeness and absence of unintended functionality can be demonstrated.

Linux conflicts with this expectation:

- The kernel contains vast quantities of implementation-driven code, developed incrementally without DAL-style requirements decomposition,
- Core subsystems (scheduler, memory management (MM), VFS, network stack, timers, softirq) lack formalized HLR/LLR specifications, making traceability impossible,
- Many kernel behaviors are emergent from implementation and concurrency rules (e.g., locking protocols, Read-Copy Update (RCU) semantics, and memory-reclaim triggering conditions). While such behaviors can be described in principle, producing a complete, stable, and reviewable DAL-style requirements decomposition is impractical. Maintaining bidirectional traceability for that decomposition across Linux's scale and ongoing evolution is also impractical.
- DO-178C code-level objectives typically require explicit specification and verification of exception and abnormal paths (e.g., defensive completion of decision logic such as handling all branches of conditionals and providing default handling for case selections). Linux follows conventional general-purpose coding practices and does not systematically enforce such complete exception-path handling across all kernel decisions; bringing the kernel into DAL A/B conformance would therefore require extensive refactoring and additional defensive logic.

To bring Linux into DAL A/B development conformance would require rewriting vast portions of the kernel under DO-178C processes—defeating the purpose of adopting Linux in the first place.

4.6.3. Verification Process Impact (Reviews, Test, MC/DC, Robustness)

DO-178C Section 6 defines verification as a technical assessment performed via a combination of reviews, analyses, and tests (DO-178C 6.0, 6.1, and 6.2). For DAL A/B, verification must provide confidence that the integrated Executable Object Code satisfies the intended requirements and that unintended functionality has been removed, including robustness with respect to abnormal inputs and conditions (DO-178C 6.1(d) and 6.1(e)). When the operating system kernel is part of the airborne software TCB, these verification expectations apply to the kernel baseline as well. Structural coverage analysis up to MC/DC at the source level,

 doi:10.20944/preprints202603.0354.v6

- Reviews and analyses of source code to confirm compliance with low-level requirements and architecture, traceability, verifiability, and correctness aspects such as memory/stack usage, exception handling, data corruption due to task/interrupt conflicts, and worst-case execution timing (WCET) considerations (DO-178C 6.3.4).
- Requirements-based testing, including robustness (abnormal-range) test cases and procedures, executed in appropriate environments (DO-178C 6.4.1, 6.4.2, and 6.4.3, especially 6.4.2.2).
- Detailed review of integration outputs (compiling/linking/loading data and memory map) and test-coverage analyses to confirm requirements-based coverage and structural coverage at the applicable level (DO-178C 6.3.5 and 6.4.4.1, 6.4.4.2).
- Resolution of uncovered or extraneous code (including dead and deactivated code) and maintenance of verification traceability across requirements, test cases, procedures, and results (Sections 6.4.4.3 and 6.5), so that unintended functionality is addressed (Section 6.1(d)).

Linux's scale makes these obligations infeasible:

- Reviews and analyses of requirements and architecture to ensure verifiability, target-computer compatibility (including interrupts/asynchronous operation), and partitioning integrity where applicable (DO-178C 6.3.1, 6.3.2, and 6.3.3).
- Dynamic kernel subsystems—such as memory reclaim, workqueues, softirq, and RCU—are driven by system-level conditions including variable load, interrupt patterns, scheduler decisions, and configuration parameters. Because these external stimuli cannot be deterministically bounded, execution timing exhibits non-deterministic variation. It follows that neither complete coverage closure nor a rigorously bounded WCET can be guaranteed under all operating conditions.
- MC/DC on the kernel would require analyzing tens of thousands of complex decision points, many interacting across subsystems.
- The Linux kernel contains a large volume of configuration-dependent, architecture-specific, or rarely executed code paths (e.g., unused error paths, debug logic, fallback handlers, and deep #ifdef branches), many of which cannot be deterministically exercised or fully covered, and often have no requirement-level justification.

The verification burden alone exceeds the cost and feasibility envelope of civil certification.

4.6.4. Configuration Management (SCMP, Baseline Control, Change Records)

For DAL A/B, DO-178C's Software Configuration Management (SCM) expectations require a defined and controlled software configuration, the ability to reproduce the Executable Object Code, disciplined baseline establishment, and traceable change control (e.g., DO-178C 7.2.2, 7.2.4, and 7.2.7). Linux conflicts with these objectives through its sheer configuration scale: modular drivers, architecture variants, and build-time feature selection collectively create an explosion of configuration items and derived artifacts that must be identified, baselined, and controlled. The resulting SCM scope is inherently difficult to bound for a monolithic, feature-rich kernel and amplifies the cost of maintaining certification evidence.

Beyond this "static" SCM scope problem, Linux also suffers from lifecycle-level baseline instability driven by continuous upstream change. That dynamic aspect—and its direct impact on re-verification and long-term certifiability—is discussed in Section 4.8.

 doi:10.20944/preprints202603.0354.v6

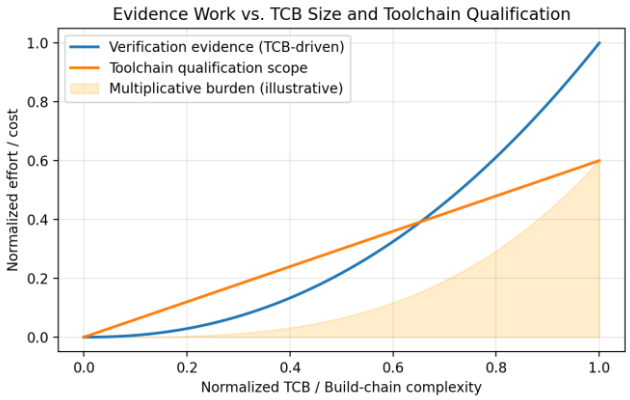


Figure 9. TCB and toolchain: the certification cost curve.

4.6.5. Quality Assurance (SQAP, Process Audits, Independence Requirements)

Under DO-178C Section 8, the Software Quality Assurance (SQA) process provides confidence that the software life cycle processes and their outputs conform to the approved plans and standards, that deficiencies are detected and tracked to resolution, and that required transition criteria are satisfied (DO-178C 8.1 and 8.2). For software submitted for certification, SQA also includes a conformity review to obtain assurance that life cycle processes and data are complete and that the delivered executable configuration is controlled and can be regenerated from archived data (DO-178C 8.3).

Linux makes these SQA expectations difficult to satisfy in a certification-grade manner because:

- Upstream kernel development is performed by a large, distributed community, so an applicant cannot realistically audit the full set of life cycle processes for plan/standard compliance or systematically manage approved deviations at the scale envisioned by SQA audits (DO-178C 8.2(d)), nor can it treat upstream change sources as controlled "suppliers" in the DO-178C sense without establishing a separate, project-owned governance and audit regime (DO-178C 8.2(i)).
- The kernel's enormous TCB makes effective independent verification and SQA oversight impractical: auditing process execution, tracking deviations and problem reports to closure, and performing a meaningful software conformity review that demonstrates completeness and regenerability of the delivered executable configuration (DO-178C 8.3) becomes prohibitively large in scope when the OS itself is part of the certified baseline.

ARINC 653-based systems take the opposite approach:

- The separation kernel is purpose-designed to be extremely small, static, and analyzable.
- Device drivers and applications run outside the certified kernel, in isolated partitions.
- The separation kernel's TCB is on the order of tens of thousands of lines, not millions, making DO-178C planning, verification, configuration control, and QA processes achievable at DAL A.

This minimal-TCB architecture exists specifically to avoid the audit explosion that characterizes monolithic kernels like Linux.

4.7. Complex Toolchain Imposes Prohibitive DO-330 Qualification Burden

Linux relies on a broad and deeply layered toolchain ecosystem that includes compilers, linkers, meta-build systems, configuration generators, device-tree compilers, package utilities, and numerous scripting tools. This ecosystem is structurally different from avionics development environments, both in scale and in the number of transformations that occur between source and final binaries.

DO-330 requires qualification for every tool that can affect airborne software, as well as for each transformation stage that generates derived artifacts. Linux's build process depends on dozens of such tools—GCC/LLVM, binutils, kbuild, Kconfig, autotools, CMake, Yocto, Device Tree (DT) compilers, Python and shell-based code generators, pkg-config, ninja, and many others. Each

 doi:10.20944/preprints202603.0354.v6

participates in multiple stages of the build pipeline, producing intermediate files, scripts, headers, configuration databases, or hardware description blobs consumed by the kernel.

Under DO-330, each tool and each interaction between tools must be justified or qualified, and the scope scales combinatorially. This creates an immense qualification envelope that is economically infeasible for DAL A/B programs.

In addition, DO-178C DAL A structural coverage guidance recognizes that compilers, linkers, and other transformation stages may introduce additional object code that is not directly traceable to source statements, in which case additional verification is required to establish correctness (Section 6.4.4.2(b)). Modern optimizing compilers and linkers routinely perform implicit transformations (inlining, vectorization, reordering, architecture-dependent instruction selection, and link-time optimization), which can expand this "non-traceable additional code" surface and increase the verification burden when attempting to apply DO-178C objectives to a large, highly optimized Linux kernel build.

Avionics RTOS environments deliberately employ minimal, deterministic, and long-term-stable toolchains. A single vendor-qualified compiler, along with a simple assembler and linker, constitutes the entire TQL surface. Build orchestration may still use a meta-build system (e.g., CMake)—including in large aerospace programs—but in avionics it is typically kept within a limited, project-selected tool environment; importantly, such tooling must still be assessed under DO-330 to determine whether it could introduce or mask errors in airborne software. The critical distinction is the absence of a distribution build stack (e.g., Yocto/Buildroot) and large transitive host-tool dependency chains. In turn, automatic code generation and distribution-level tool dependencies are minimized or eliminated.

4.8. Continuous Patch Stream Destabilizes Certified Baselines

Linux evolves at a rapid pace, and maintaining a stable, certifiable baseline is structurally incompatible with this development model. Kernel updates are continuous because Linux must support a vast hardware ecosystem, address frequent security vulnerabilities, and resolve behavioral regressions across numerous subsystems. Importantly, these changes are not limited to peripheral drivers; they routinely modify core kernel semantics and internal interfaces that directly affect timing behavior, memory management, and fault-propagation pathways. In practical certification terms, any baseline change—no matter how small—can invalidate previously accepted verification evidence and trigger change control and re-verification activities: impact analysis, regression testing, artifact updates, and re-establishment of traceability between requirements, code, tests, and results. When the baseline itself is forced to move continuously, the program enters a re-verification spiral in which maintaining long-term, archived, and releasable configurations becomes increasingly difficult (cf. the DO-178C SCM expectations discussed in Section 4.6.4). The situation is worse for PREEMPT_RT, the component most often cited to justify Linux for real-time avionics. PREEMPT_RT remains under active refinement (e.g., threaded interrupt behavior, sleeping spinlocks, priority inheritance details, and scheduler/locking interactions), so upstream changes can alter latency distributions and even the correctness assumptions of timing analysis. Freezing a PREEMPT_RT-based kernel therefore quickly diverges from upstream and demands sustained maintenance and repeated re-verification—precisely in the kernel areas that would need the strongest stability for DAL A/B timing evidence.

On top of that, Linux's upstream software configuration management (SCM) workflow is built around general-purpose tooling (e.g., Git-based change control and release processes) that is not delivered as a DO-330-qualified tool environment. Therefore, any attempt to treat mainline Linux + upstream change workflow as part of a certification baseline would either require a program-specific DO-330 tool-qualification effort for the complete change-and-release toolchain, or else require additional independent verification controls to compensate for unqualified tool outputs.

 doi:10.20944/preprints202603.0354.v6

5. Unified Causal Structure of Linux's Certification Challenges

This work is summarized through a unified causal chain that explains how Linux's general-purpose design choices can translate into certification-critical challenges in safety-critical avionics contexts.

Root Cause: Linux is a function-rich, high-performance general-purpose operating system. This design choice inherently triggers two downstream consequences, forming a coherent causal chain that explains Linux's incompatibility with safety-critical avionics:

5.1. Airworthiness Infeasibility

Consequences Derived from Airworthiness Infeasibility. The first consequence is attributable to two additional certifiability-blocking properties:

5.1.1. An Excessively Large Trusted Computing Base (TCB)

5.1.2. A Highly Complex Toolchain That Imposes Prohibitive DO-330 Qualification Burdens

Jointly Caused Consequence. Finally, both primary branches—airworthiness infeasibility and semantic complexity—jointly produce:

5.1.3. Continuous Patch-Stream Evolution

5.2. Complex and Open System Semantics

The execution state space becomes non-finite, workload-dependent, and continuously evolving.

The second primary consequence Derived from two major technical deficiencies:

5.2.1. Temporal Non-Isolation – Arising from Asynchronous Kernel Activity, Dynamic Scheduling Behavior, and Non-Preemptible Regions

5.2.2. Spatial Non-Isolation – Further Decomposing into Three Architectural Mechanisms:

- Mutable logical-to-physical memory mappings,
- Driver-induced contamination of globally shared kernel state,
- Lack of enforced fault-containment boundaries.

Consequently, this causal chain provides a complete and coherent explanation for all eight architectural deficiencies discussed in this paper, showing that they are not isolated issues but systemic consequences of Linux's foundational design philosophy.

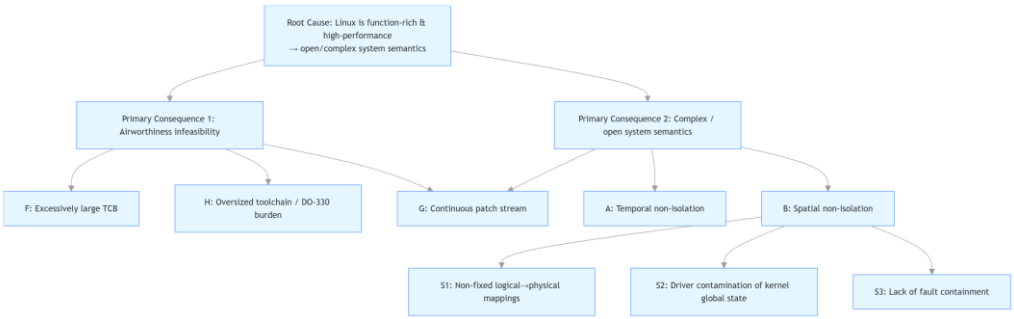


Figure 10. Two primary consequences and eight derived architectural deficiencies forming the Linux certification risk structure.

 doi:10.20944/preprints202603.0354.v6

6. Common Misunderstandings and Why They Fail

6.1. PREEMPT_RT = Determinism

6.1.1. Misconception

Applying PREEMPT_RT makes Linux suitable for DAL A/B airborne workloads with controlled timing interference and deterministic CPU-time allocation.

6.1.2. Reality

PREEMPT_RT improves average-case latency by threading interrupts and shrinking non-preemptible regions, but it does not turn Linux into a temporally deterministic system.

Unbounded interference sources remain:

- IRQ exit softirq cascades;
- memory management events (page faults, reclaim, compaction)
- cross-CPU TLB shootdowns that can outrank RT tasks;
- driver execution time that depends on firmware/DMA completion/lock contention;
- DVFS and CPU C-state exits that introduce microarchitectural stalls outside scheduler control;
- dynamic scheduler behavior (wakeup rules, load balancing, task migration, housekeeping threads) that produces runtime-dependent jitter.

Because of these, even a high-priority real-time task may fail to preempt and run immediately when it becomes runnable (dispatch/preemption latency), and may be preempted or lose CPU time while executing.

6.1.3. Avionics Impact

DAL A/B avionics needs temporal determinism backed by analyzable upper bounds on interference. ARINC 653 achieves this with fixed, table-driven execution windows (temporal partitioning). Linux (even with PREEMPT_RT) remains an as-soon-as-possible (ASAP) scheduling model, in which asynchronous kernel activities can consume CPU time unpredictably. Therefore, PREEMPT_RT cannot support avionics-grade temporal partitioning or provide partition-level schedulability evidence under DO-178C.

6.2. Containers/ Virtual Machines (VMs)/Cgroups = Partitioning

6.2.1. Misconception

Namespaces/Containers/Cgroups/VMs (on a Linux host) provide avionics-grade partitioning equivalent to ARINC 653.

6.2.2. Reality

Containers and Linux hosted VMs ultimately share the same monolithic kernel and its global state. Cgroups offer resource shaping, not hardware enforced temporal or spatial isolation. As long as kernel global data structures, interrupt domains, MM events, and driver paths are shared, faults and timing interference can propagate.

In essence, these mechanisms still amount to application-level constrained or bound processes (or process groups), and therefore retain the composition-level fault-contagion risk described in Sections 4.2, 4.3, and 4.5.

 doi:10.20944/preprints202603.0354.v6

6.2.3. Avionics Impact

ARINC 653 requires strict temporal partitioning (fixed, table driven major/minor frames) and strict spatial isolation (hardware enforced, immutable physical memory regions). Cgroups provide neither:

- Enforcing strict and deterministic execution budgets, the cgroup scheduler provides only proportional CPU time shares. At its core, each group receives an average amount of runtime but with no guarantee of bounded latency or deterministic execution timing.
- Spatial isolation fails because cgroups do not define exclusive physical memory ranges, do not prevent page migration, compaction, NUMA balancing, or TLB shootdowns, and do not protect kernel global state.
- Fault isolation fails because any cgroup can corrupt shared kernel global data structures or destabilize drivers, affecting others. Therefore, cgroups/containers/VMs cannot satisfy DO-178C DAL A/B isolation.

Table 1. Linux cgroups vs ARINC 653 partitions.

Aspect	Linux cgroups	ARINC 653 Partitions
Isolation model	Resource quotas (CPU/mem/IO); no hard isolation	Hardware enforced spatial isolation (MMU)
Kernel domain	All tasks share one monolithic kernel	Separation kernel enforces strict boundaries
Fault containment	None – faults propagate through shared kernel state	Strong – faults confined to partition
Memory separation	No exclusive physical regions; no immutable mappings	Dedicated address spaces, fixed at integration
Time isolation	No deterministic execution or WCET guarantees	Deterministic major/minor frame scheduling
System semantics	Dynamic, open world, non enumerable	Closed, analyzable, integration time frozen
TCB size	Millions of LOC (grows with every patch)	Small, static, certifiable separation kernel
Certification suitability	Cannot meet DO 178C DAL A/B	Designed specifically for DAL A/B compliance
Purpose	Resource management	Safety critical partitioning architecture
Nature	Policy mechanism inside Linux	Architectural foundation of safety systems

6.3. Using mlock() and Disabling Swap to Fix Partition Memory

6.3.1. Misconception

Locking pages with mlock() or disabling swap provides deterministic memory behavior and partition-like physical isolation.

6.3.2. Reality

Both mechanisms only affect paging, not physical-memory determinism. Even with swap disabled and pages locked, Linux may still reclaim, migrate, compact, or remap physical pages, and continues to update Page Table Entries (PTE) and trigger TLB shootdowns. Memory remains part of a shared, dynamic global pool—not a statically isolated region.

6.3.3. Avionics Impact

For DAL A/B certification under DO-178C, the platform must provide robust, hardware enforced partition isolation, which in practice depends on establishing stable, non-bypassable memory-protection boundaries throughout operation. Linux's dynamic memory subsystem prevents

 doi:10.20944/preprints202603.0354.v6

establishing such static boundaries (as discussed in Section 4.3), making spatial isolation infeasible under DO-178C DAL A/B.

6.4. Static Configuration Make Linux Deterministic

6.4.1. Misconception

Fixing the process/thread set, statically allocating stacks and memory regions, constraining communication patterns, masking interrupts, removing signals, or otherwise "freezing" the application-level structure can transform Linux into a closed, deterministic execution environment.

6.4.2. Reality

Static application level configuration can reduce user space variability (e.g., by fixing the process/thread inventory, stack sizes, memory budgets, and IPC topology), but it does not change Linux's open world execution semantics. The kernel still performs autonomous, runtime-dependent activities driven by interrupts, device/driver behavior, memory pressure, and subsystem heuristics. Hence, new execution paths and interleavings continue to emerge during operation and cannot be completely enumerated or frozen at integration time (Section 4.1).

6.4.3. Avionics Impact

Static configuration cannot, by itself, establish the partitioning and schedulability assurances required for certification, because Linux's kernel behavior remains runtime-driven and non-enumerable (Sections 4.1, 4.2, and 4.3).

6.5. Abundant Linux Ecosystem

6.5.1. Misconception

Linux's extensive ecosystem—its diverse toolchains, build systems, subsystems, and libraries—provides engineering convenience and therefore should make it easier to build safety-critical airborne software.

6.5.2. Reality

The breadth of the Linux ecosystem increases engineering convenience but dramatically amplifies certification complexity. Linux depends on a large and heterogeneous collection of compilers, linkers, meta-build systems, configuration generators, device-tree compilers, packaging tools, scripting environments, and subsystem-specific utilities. Each of these components introduces additional transformations, dependencies, and versioned artifacts that must be assessed under DO-330 when they affect airborne software.

6.5.3. Avionics Impact

For DAL A/B avionics, "more tools and libraries" usually means more certification scope: more tool qualification exposure under DO-330, more configuration items to control, and more change impact to re-verify. As analyzed in Section 4.8, ecosystem richness tends to increase assurance cost and lifecycle risk, even if it reduces prototype effort. Thus, the richness of the Linux ecosystem—an advantage for general engineering—becomes a certification burden under DO-178C/DO-330 and is misaligned with the lifecycle stability expected in airborne systems.

 doi:10.20944/preprints202603.0354.v6

6.6. Open Source = Reduces Cost

6.6.1. Misconception

Because Linux is open source and has no licensing fees, adopting it should reduce overall program cost for airborne software.

6.6.2. Reality

Licensing is usually not the dominant cost driver in a DAL A/B DO-178C program. What dominates is the amount of assurance work and evidence that must be planned, traced, verified, qualified, configuration-controlled, and repeatedly re-verified over the lifecycle. A general purpose, fast evolving, heterogeneous Linux stack expands the Trusted Computing Base (TCB) and the volume of derived artifacts and dependencies, leading to higher certification cost and schedule risk..

6.6.3. Avionics Impact

Under DO-178C, certification cost is driven not by software licensing fees but by the scale and stability of the assurance evidence required. Cost grows with:

- Evidence volume across the full chain of requirements, design, code, verification, coverage, and change-impact artifacts.
- Independence and quality-assurance activities including Quality Assurance/Independent Verification and Validation (QA/IV&V) needed to satisfy DAL A/B objectives.
- Supplier and version control, including long-term reproducibility, auditability, and traceability of all build inputs.

These factors increase certification burden rather than reducing cost, despite the absence of licensing fees.

Table 2. Common misconceptions about Linux for safety-critical systems.

Misconception	Concise Refutation	Section
PREEMPT_RT ⇒ determinism	Improves latency; cannot bound worst-case timing	**§6.1**
Cgroups/VMs ⇒ partitions	CPU share only; no fixed windows or isolation	**§6.2**
mlock()/no-swap ⇒ isolation	Residency ≠ exclusivity; mappings still change	**§6.3**
Static config ⇒ deterministic	Static layout ≠ static semantics; kernel remains dynamic	**§6.4**
Rich ecosystem ⇒ cert-friendly	Toolchain heterogeneity breaks DO-330 traceability	**§6.5**
Open source ⇒ reduces cost	Certification effort scales with verification scope, not license fee	**§6.6**

As shown above, Linux's unsuitability for DAL A/B cannot be "engineered away" through incremental hardening, configuration tightening, or other minor-to-moderate kernel modifications; the architectural blockers identified in this paper remain. If changes are made extensive enough to satisfy avionics operating-system assumptions (closed-world semantics, analyzable partition-level timing, and provable spatial/fault isolation), the result ceases to be meaningfully "Linux" in the COTS/upstream sense and becomes a totally different operating system with a new certification burden.

 doi:10.20944/preprints202603.0354.v6

7. Practical Implications for Avionics OS Selection

This section summarizes practical implications derived directly from the refutation and risk-factor analysis in Sections 4–6. A comprehensive safety-governance framework (e.g., organizational processes, supplier governance, and regulator–applicant interaction models) is outside the scope of this paper.

Accordingly, we provide a small set of takeaways for teams evaluating OS/platform choices for airborne systems, emphasizing decisions that most directly affect certifiability, evidence stability, and system-level isolation.

7.1. Prefer Partitioned Architectures Over General-Purpose OSes

Safety-critical avionics require closed-world execution semantics, strong spatial separation, and bounded temporal behavior. Organizations should therefore adopt one of the following architectural classes, all of which are engineered to satisfy ARINC 653 and DO-178C principles:

7.1.1. ARINC 653 Partitioning Kernels / Type-1 Hypervisors

Suitable platforms include:

- XtratuM + LithOS
- RTEMS + AIR
- POK
- JetOS

These systems provide:

- Table-driven major/minor-frame scheduling
- Hardware-enforced spatial isolation
- Minimal separation-kernel TCB
- Deterministic inter-partition IPC

7.1.2. High-Assurance Separation Kernels with Userspace ARINC Services

Preferred when formal verification or ultra-small TCB is needed:

- Muen SK

The architectures offer:

- Minimal trusted computing base (order of 10–20 KLOC)
- Strict capability-based authority
- Provable fault isolation
- Support for deterministic mixed-criticality scheduling

7.1.3. Commercial Certifiable RTOS Platforms

When product maturity, vendor support, and certification packages are required:

- VxWorks 653
- INTEGRITY-178B
- LynxOS-178
- PikeOS
- DeOS

These products include:

- Controlled and frozen baselines
- Vendor-qualified toolchains
- Well-established certification artifacts

 doi:10.20944/preprints202603.0354.v6

- Deterministic partitioning services

7.2. Enforce Minimal Trusted Computing Base (TCB) as a Primary Design Objective

From earlier analysis, Linux's monolithic TCB is a certifiability blocker. Future airborne systems should adopt the following TCB-reduction strategies:

- Keep only scheduling, memory isolation, and IPC in the trusted kernel.
- Push device drivers and services to unprivileged partitions; use IOMMU/MPU/VT-d mechanisms to sandbox drivers in their own partitions.
- Avoid shared kernel global state and shared writable memory: the platform architecture must enforce non-transitive fault containment.
- Ensure deterministic and static memory mappings: no demand paging, no dynamic reclaim, no runtime compaction, no page migration.

A minimal, analyzable TCB is the single most effective way to reduce certification cost and risk.

7.3. Minimal Lifecycle Considerations

The following lifecycle considerations help interpret the practical feasibility of the mitigation narratives discussed in Section 6 and the type of evidence typically needed to sustain them over an airborne lifecycle.

- Enforce architectural separation of DAL A/B
- Require early airworthiness review for platform architecture selection (partitioning kernel/hypervisor, driver model)
- Document OS-selection rationale in the system PSAC artifacts
- Additionally, treat toolchain choice, baseline control, and assurance competence as governance items rather than ad-hoc engineering decisions:
- Adopt a narrow, stable, vendor-qualified toolchain.
- Freeze baselines early and maintain long-term configuration control.
- Develop organizational capability in standards-based assurance.

Overall, these recommendations aim to reduce certification risk by making critical architectural and lifecycle decisions explicit and enforceable.

8. Conclusions

This work analyzed how Linux's architecture and lifecycle model interact with assurance objectives commonly applied to DAL A/B avionics. Architecturally, Linux retains open-world execution semantics (Section 4.1), exhibits timing variability that complicates analyzable temporal isolation (Section 4.2), and employs mutable memory-management mechanisms that challenge establishing immutable, hardware-enforced partition memory regions in the sense typically assumed by partitioned avionics platforms (Section 4.3). In addition, fault-containment boundaries are difficult to demonstrate at the system level when failures can propagate through shared kernel state and privileged driver execution (Sections 4.4 and 4.5). These technical properties, together with lifecycle factors such as baseline stability (Sections 4.6.4 and 4.8), development and traceability practices at scale (Section 4.6.2), tool qualification exposure (Section 4.7), and the feasibility of producing certification evidence across planning, verification, configuration management, and quality assurance (Sections 4.6.1–4.6.5), collectively indicate substantial certification risk for Linux-based approaches in flight-critical DAL A/B contexts.

Linux remains an effective platform for prototyping and offers practical advantages that are attractive to new entrants in emerging aviation domains. However, for flight-critical functions at DAL A/B, the analysis suggests that relying on Linux as the primary execution foundation can create assurance and lifecycle risks that are difficult to control within conventional DO-178C/DO-330 certification expectations. As low-altitude aviation continues to expand and attract cross-industry

 doi:10.20944/preprints202603.0354.v6

participants, OS and platform-architecture decisions benefit from early, explicit alignment with partitioning, determinism, isolation, and evidence-stability objectives so that long-term safety assurance is not compromised by short-term development convenience.

References

1. ARINC Industry Activities, *ARINC Specification 653P1-3: Avionics Application Software Standard Interface, Part 1*, Annapolis, MD, USA: ARINC, 2015.
2. ARINC Industry Activities, *ARINC Specification 653P3-2: Avionics Application Software Standard Interface, Part 3*, Annapolis, MD, USA: ARINC, 2014.
3. RTCA Inc., *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*, Washington, DC, USA: RTCA, 2011.
4. EUROCAE, *ED-12C: Software Considerations in Airborne Systems and Equipment Certification*, Paris, France: EUROCAE, 2011.
5. RTCA Inc., *DO-297: Integrated Modular Avionics (IMA) Design Guidance and Certification Considerations*, Washington, DC, USA: RTCA, 2005.
6. RTCA Inc., *DO-330: Software Tool Qualification Considerations*, Washington, DC, USA: RTCA, 2011.
7. SAE International, ARP4754A: Guidelines for Development of Civil Aircraft and Systems, Warrendale, PA, USA: SAE, 2010.
8. P. Wang, Q. Li, & H. Xiong, "Time and space partitioning technology for integrated modular avionics systems," *J. Beijing Univ. Aeronaut. Astronaut.*, vol. 38, no. 6, pp. 721–726, 2012 (in Chinese).
9. F. He, H. Xiong, & X. Zhou, "Overview of key technologies for ARINC 653 partitioned operating systems," *Acta Aeronaut. Astronaut. Sin.*, vol. 35, no. 7, pp. 1777–1796, 2014 (in Chinese).
10. Y. Li, T. Zhou, & J. Li, "Research and implementation of airborne ARINC 653 partition operating system," *Comput. Eng. Appl.*, vol. 51, no. 20, pp. 235–240, 2015 (in Chinese).
11. L. Chen, "Research on deterministic scheduling of avionics partition operating systems," Ph.D. dissertation, Coll. Aeronaut. Eng., Nanjing Univ. Aeronaut. Astronaut., Nanjing, China, 2018 (in Chinese).
12. R. Huang, "Research on ARINC 653 partition isolation mechanism for IMA," Ph.D. dissertation, Sch. Electr. Eng., Northwestern Polytech. Univ., Xi'an, China, 2020 (in Chinese).
13. I. Lopez, P. Parra, M. Urueña, et al., "XtratuM: a hypervisor for partitioned embedded real-time systems," in *Proc. 18th Int. Conf. Real-Time Netw. Syst. (RTNS)*, Paris, France: ACM, 2010, pp. 1–6.
14. A. Crespo, P. Metge, & I. Lopez, *LithOS: A Guest OS for ARINC 653 on XtratuM Hypervisor*, Valencia, Spain: Univ. Politèc. Valencia, 2012.
15. J. Delange, L. Pautet, & S. Faucou, "POK: an ARINC 653 compliant operating system for high-integrity systems," in *Reliable Software Technologies – Ada-Europe 2010*, Berlin, Germany: Springer, 2010, pp. 172–185.
16. B. Huber, A. Lackorzynski, A. Warg, et al., "seL4: formal verification of a high-assurance microkernel," *Commun. ACM*, vol. 57, no. 3, pp. 107–115, 2014.
17. I. Kuz, K. Elphinstone, G. Heiser, et al., "MCS: temporal isolation in the seL4 microkernel," in *Proc. 11th Oper. Syst. Platforms Embedded Real-Time Appl. (OSPERT)*, New York, NY, USA: IEEE, 2015, pp. 1–6.
18. H. Härtig, A. Lackorzynski, & A. Warg, *The Muen Separation Kernel: Design and Formal Verification*, Dresden, Germany: Tech. Univ. Dresden, 2018.
19. J. Rushby, *Design and Verification of Secure Systems*, Menlo Park, CA, USA: SRI Int., 1981.
20. J. Rushby, "A kernelized architecture for safety-critical systems," in *Proc. IFIP Congr.*, Vienna, Austria, 1999, pp. 1–6.
21. Wind River Systems Inc., *VxWorks 653 Platform Datasheet*, [Online]. Available: https://www.windriver.com, 2022.
22. Green Hills Software Inc., *INTEGRITY-178B RTOS for Avionics*, [Online]. Available: https://www.ghs.com, 2021.
23. SYSGO AG, *PikeOS Safety-Certifiable RTOS and Hypervisor*, [Online]. Available: https://www.sysgo.com, 2024.
24. DDC-I Inc., *DeOS Safety-Critical RTOS*, [Online]. Available: https://www.ddci.com, 2024.
25. D. Bovet, & M. Cesati, *Understanding the Linux Kernel*, Sebastopol, CA, USA: O'Reilly Media, 2005.

 doi:10.20944/preprints202603.0354.v6

26. R. Love, *Linux Kernel Development*, Upper Saddle River, NJ, USA: Addison-Wesley, 2010.
27. M. Gorman, *Understanding the Linux Virtual Memory Manager*, Upper Saddle River, NJ, USA: Prentice Hall, 2004.
28. The Linux Kernel Organization, *Linux Scheduler Documentation*, [Online]. Available: https://docs.kernel.org/scheduler/, 2024.
29. The Linux Kernel Organization, *Linux Memory Management Documentation*, [Online]. Available: https://docs.kernel.org/mm/, 2024.
30. T. Gleixner, *PREEMPT_RT Patch Overview and Design Philosophy*, San Francisco, CA, USA: Linux Foundation, 2019, [Online]. Available: https://wiki.linuxfoundation.org/realtime/start.
31. The Linux Kernel Organization, *kbuild: The Linux Kernel Build System*, [Online]. Available: https://docs.kernel.org/kbuild/, 2024.
32. The Yocto Project, *Yocto Project Mega-Manual*, [Online]. Available: https://www.yoctoproject.org, 2024.
33. Device Tree Working Group, *Device Tree Specification*, [Online]. Available: https://www.devicetree.org, 2024.
34. H. Zhao, S. Gao, & Y. Yang, "Applicability analysis of airborne software based on Linux real-time extension," *Comput. Eng.*, vol. 43, no. S1, pp. 311–315, 2017.
35. a653rs Contributors, *a653rs-linux: ARINC 653 Emulation on Linux*, [Online]. Available: https://github.com/a653rs, 2024.
36. ELISA Project, *FAQs – ELISA: Enabling Linux in Safety Applications*, [Online]. Available: https://elisa.tech/about/faqs/, Accessed: 2026-03-23.